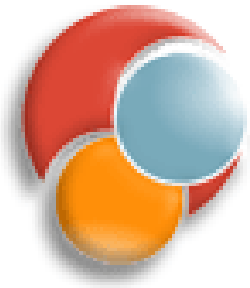


Java y Herramientas de Desarrollo

Sesión 9: Consultas a una BD con JDBC

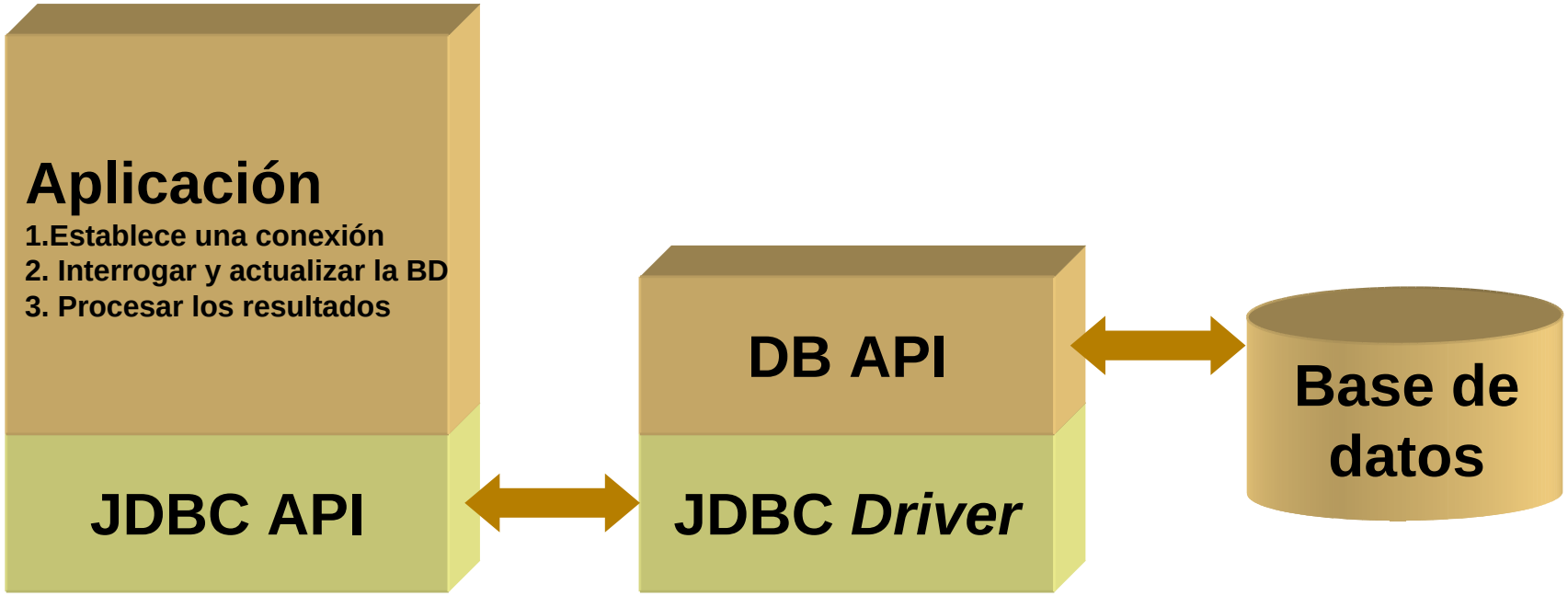
Puntos a tratar

- Introducción
- *Drivers* de acceso a bases de datos
- Conexión con la base de datos
- Consulta a una base de datos
- Restricciones y movimientos en el *ResultSet*
- Sentencias de actualización
- Otras llamadas a la BD

Introducción

- JDBC es el API de Java para acceder a sistemas de gestión de bases de datos (SGBD)
- Al hacer uso del API nos va a permitir cambiar de SGBD sin modificar nuestro código
- JDBC es una especie de “puente” entre nuestro programa Java y el SGBD

Esquema de uso de JDBC

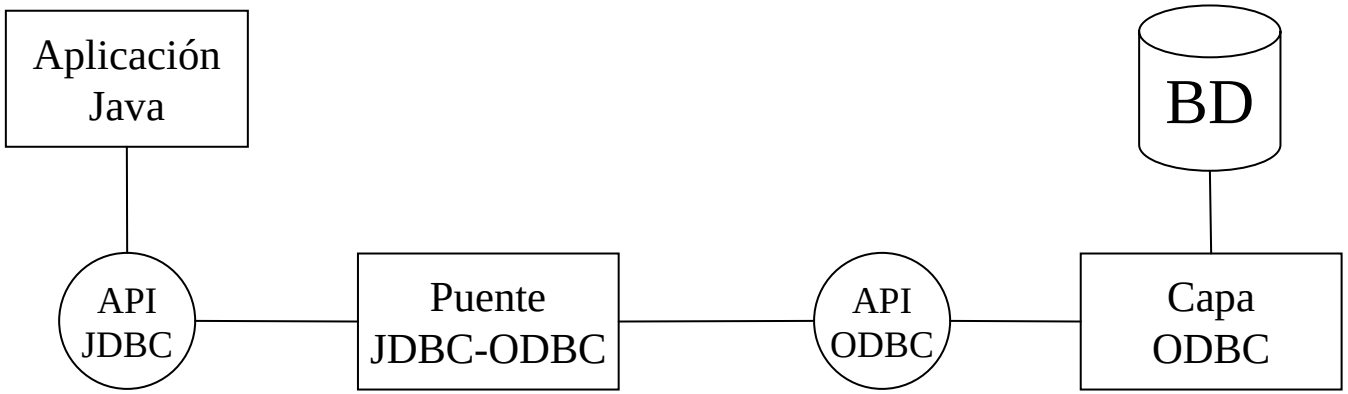


Drivers de acceso

- Para acceder a una BD necesitamos un *driver* específico
- Cada BD suele disponer de un API de acceso propietario
- Si usamos ese API, un cambio en la BD provocaría cambios en nuestro código
- El *driver* es específico para esa BD, al cambiar la BD sólo tenemos que cambiar el *driver*
- El *driver* traduce la llamada JDBC en la correspondiente llamada al API de la BD

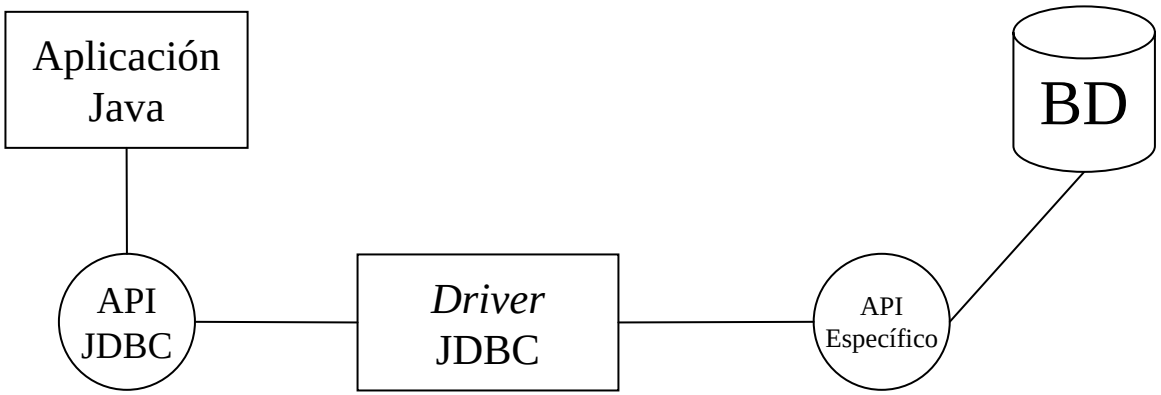
Tipos de *drivers*

- Tipo 1: Puente JDBC-ODBC
  - Proporciona conectividad entre Java y cualquier base de datos en Microsoft Windows, mediante ODBC
  - No se aconseja su uso. Limita las funcionalidades de las BD
  - Cada cliente debe tener instalado el *driver*
  - J2SE incluye por defecto este *driver* (Windows y Solaris)



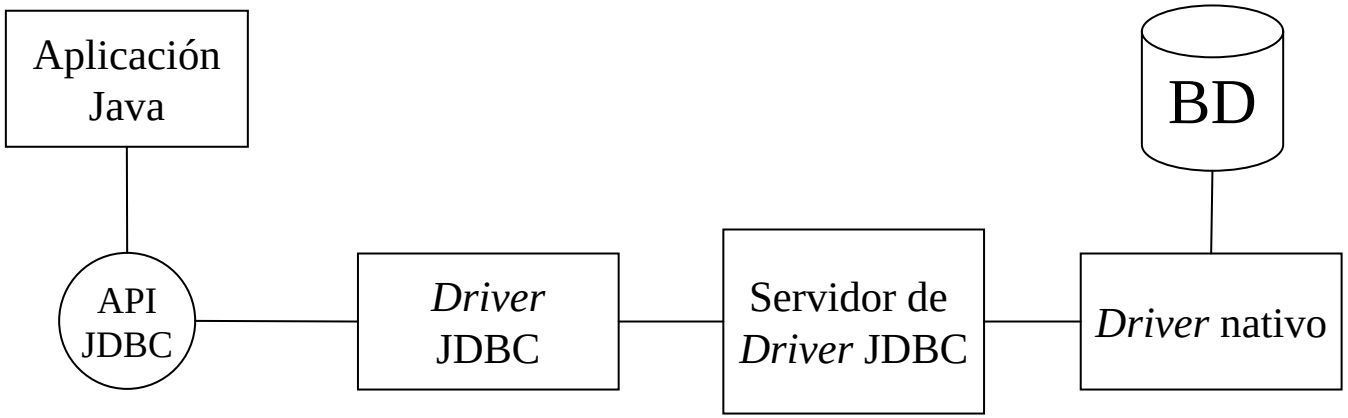
Tipos de *drivers*

- Tipo 2: Parte Java, parte *driver* nativo
  - El *driver* actúa como traductor de la llamada Java a una llamada del API de la BD. Necesita el API de forma local (no usar en Internet)
  - Es un paso menos que el anterior, pues no tenemos que pasar por el gestor ODBC (más rápido)
  - Cada cliente necesita el *driver*



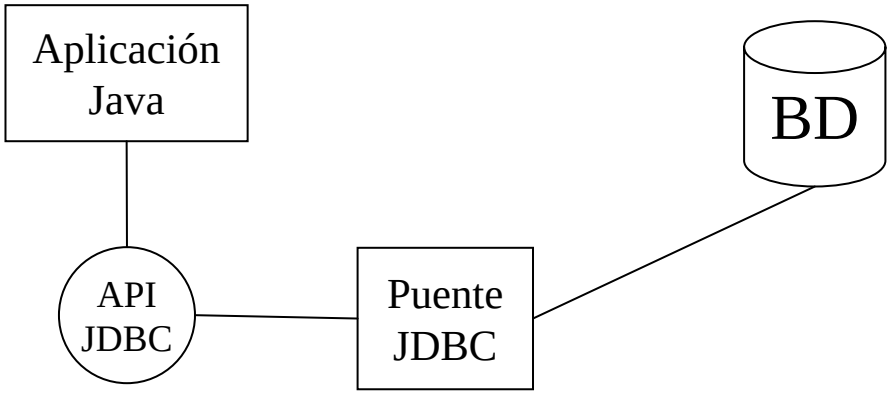
Tipos de *drivers*

- Tipo 3: Servidor intermediario de acceso a BD
  - Proporciona una mayor abstracción
  - Dispondremos de un componente servidor intermedio, que gestiona la conexión con una o varias BD
  - WebLogic implementa este *driver*
  - Útil para aplicaciones escalables y portables



Tipos de *drivers*

- Tipo 4: *Drivers* Java
  - El más directo
  - La llamada JDBC se traduce en una llamada a la propia BD, por la red y sin intermediarios
  - Mejor rendimiento
  - La mayoría de SGBD disponen de este *driver*



Sobre los distintos tipos

- Podemos disponer de *drivers* de distinto tipo para acceder a la misma BD
- Por ejemplo, MySQL desde su propio *driver* y desde ODBC
- Debemos tener en cuenta que un tipo de *driver* puede limitar las funcionalidades de la BD. En este caso, si utilizamos ODBC no tendremos acceso al control de transacciones de MySQL
- Resumiendo, utilizar siempre el *driver* del fabricante

Instalación de *drivers*

- Descargamos el *driver* específico para nuestra BD (normalmente es un .jar)
- Lo añadimos al CLASSPATH

```
export CLASSPATH=$CLASSPATH:/dir-donde-este/fichero
```

- Lo cargamos de forma dinámica dentro de nuestro código Java:
  - MySQL: `Class.forName("com.mysql.jdbc.Driver");`
    Podéis encontrar también la clase `org.gjt.mm.mysql.Driver`
  - PostGres: `Class.forName("org.postgresql.Driver");`
  - ODBC: `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`
- Se deben capturar las excepciones `SQLException`

Conexión a la BD

- Primero debemos conectarnos con la BD

```
Connection con = DriverManager.getConnection(url);
Connection con = DriverManager.getConnection(url,
                                             login, password);
```

- El url cambiará de una BD a otra, pero todas mantendrán el siguiente formato:
  - `jdbc:<subprotocolo>:<nombre>`
  - *jdbc* siempre
  - *subprotocolo* es el protocolo a utilizar.
  - *nombre* es la dirección (o el nombre) de la BD

Ejemplos de conexiones

- MySQL

```
Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost/bd", "miguel",
  "m++24");
```

- PostGres

```
Connection con = DriverManager.getConnection(
  "jdbc:postgresql://localhost:5432/bd",
  "miguel", "m++24");
```

- ODBC

```
Connection con =
  DriverManager.getConnection("jdbc:odbc:bd");
```

DriverManager

- Este objeto gestiona todo el paso de información con el driver
- Algunos métodos útiles de esta clase:

```
DriverManager.setLogWriter(new
  PrintWriter(System.out, true)); // Muestra
```
por la salida estándar cualquier operación que se realice con el driver

```
DriverManager.println("Esto es un
  mensaje"); // Nos permite depurar nuestro código
```

Consulta a una BD

- La conexión a la BD la podemos utilizar para consultar, insertar o borrar datos
- Todas estas operaciones se realizarán mediante SQL
- La clase `Statement` nos permitirá realizar estas acciones
- Para crear un objeto de esta clase

```
Statement stmt = con.createStatement();
```

Consulta (*Query*)

- Para consultar datos utilizamos el método `executeQuery` de la clase `Statement`

```
ResultSet result =
  stmt.executeQuery(query);
```

- `query` es un `String` que contiene la sentencia SQL
- La llamada al método nos devuelve un objeto de la clase `ResultSet`
- La respuesta es una tabla que contendrá una serie de campos y unos registros, dependiendo de la consulta realizada

Ejemplo de consulta

```
String query = "SELECT * FROM ALUMNOS
  WHERE sexo = 'M'";
ResultSet result =
  stmt.executeQuery(query);
```

- Imaginemos que la tabla ALUMNOS tiene tres campos, el resultado almacenado en *result* es

Registro

exp	nombre	sexo
1286	**Amparo**	**M**
1287	**Manuela**	**M**
1288	**Lucrecia**	**M**

Acceso a los valores de `ResultSet`

- La clase `ResultSet` dispone de un cursor que nos permite movernos por los registros
- Cuando ejecutamos la llamada, el cursor está en la posición anterior al primer registro
- Para mover el cursor a la siguiente posición utilizaremos el método `next` de `ResultSet`
- `next` devuelve cierto si ha conseguido pasar al siguiente registro y falso si se encuentra en el último
- Para acceder a los datos del `ResultSet`, haremos un bucle como este:

```
while(result.next()) {
 // Leer registro
}
```

Obtención del valor de los campos

- El cursor está situado en un campo
- Para obtener los valores de los campos utilizaremos los métodos `getXXXX(campo)` donde `XXXX` es el tipo de datos Java de retorno
- El tipo de datos del campo debe ser convertible al tipo de datos Java especificado
- El campo se especifica mediante un `String` o mediante un índice entero, cuyo valor dependerá de la consulta realizada

Tipos de datos

- Los principales métodos que podemos utilizar son:

`getInt`	**Datos enteros**
`getDouble`	**Datos reales**
`getBoolean`	**Campos booleanos (si/no)**
`getString`	**Campos de texto**
`getDate`	**Tipo fecha (devuelve `Date`)**
`getTime`	**Tipo fecha (devuelve `Time`)**

Ejemplo

```
int exp;
String nombre;
String sexo;

while(result.next()) {
    exp = result.getInt("exp");
    nombre =
          result.getString("nombre");
    sexo = result.getString("sexo");
    System.out.println(exp + "\t" +
          nombre + "\t" + sexo);
}
```

Posible problema

- Si el campo a consultar no contiene ningún valor, la llamada a `get` devuelve `0`, si es número, y `null` si es un objeto
- En el caso de `nombre` si intentamos realizar una llamada a algún método de la clase `String`, se producirá una excepción
- Para evitar esto, podemos llamar al método `wasNull()`, que devuelve cierto si el último registro consultado no tenía un valor asignado

Ejemplo

```
String sexo;

while(result.next()) {
    exp = result.getInt("exp");
    System.out.print(exp + "\t");
    nombre = result.getString("nombre");
    if (result.wasNull())
        System.out.print(
                "Sin nombre asignado");
    else
        System.out.print(nombre.trim());
    sexo = result.getString("sexo");
    System.out.println("\t" + sexo);
}
```

Restricciones en la llamada

- Cuando interrogamos una BD, el resultado devuelto puede ser extremadamente grande
- Podemos limitar el número de registros a devolver
- Disponemos de dos métodos en la clase `Statement`, `getMaxRows` y `setMaxRows` que nos devuelve y cambia el máximo número de filas
- Por defecto está a 0 (no hay restricciones)
- Si cambiamos el valor (p.e. 30), una consulta no devolverá más de 30 registros

Movimientos en el `ResultSet`

- Hasta ahora hemos utilizado el método `next` para movernos por el `ResultSet`
- Podemos crear un `ResultSet` arrastable que nos permita movernos de forma no lineal
- Primero tenemos que crear un objeto `Statement` de la siguiente manera:

```
Statement createStatement (
                int resultSetType,
                int resultSetConcurrency)
```

Valores de `resultSetType`

- `ResultSet.TYPE_FORWARD_ONLY` Valor por defecto. Sólo permite el desplazamiento hacia delante
- `ResultSet.TYPE_SCROLL_INSENSITIVE` Permite el desplazamiento. Si se cambian los datos que estamos visualizando en la BD, los datos mostrados no cambian
- `ResultSet.TYPE_SCROLL_SENSITIVE` Permite el desplazamiento y cualquier cambio en la BD afecta a los datos visualizados
- El mostrar los cambios que se produzcan en la BD dependerá de la BD y del *driver* que estemos utilizando

Valores de `resultSetConcurrency`

- `ResultSet.CONCUR_READ_ONLY` Valor por defecto. Cualquier cambio en el `ResultSet` no tiene efecto en la BD
- `ResultSet.CONCUR_UPDATABLE` Permite que los cambios efectuados en el `ResultSet` tengan efecto en la BD
- Para poder actualizar los datos de la BD debemos crear un `Statement` con, al menos, `TYPE_FORWARD_SENSITIVE` y `CONCUR_UPDATABLE`

Movimientos en el `ResultSet`

- Una vez realizada la consulta y obtenido el `ResultSet` arrastable, podemos usar:

`next`	**Pasa a la siguiente fila**
`previous`	**Ídem fila anterior**
`last`	**Ídem última fila**
`first`	**Ídem primera fila**
`absolute(int fila)`	**Pasa a la fila número fila**
`relative(int fila)`	**Pasa a la fila número fila desde la actual**
`getRow`	**Devuelve la número de fila actual**
`isLast`	**Devuelve si la fila actual es la última**
`isFirst`	**Ídem la primera**

Modificación del `ResultSet`

- Para modificar un campo del registro actual, usaremos `updateXXXX` (igual que `getXXXX`)
- `updateXXXX` recibe dos parámetros, nombre del campo a modificar y nuevo valor del campo
- Para que los cambios tengan efecto debemos llamar a `updateRow`

```
rs.updateString("nombre","manolito");
rs.updateRow();
```

Modificación del `ResultSet`

- Para desechar los cambios del registro actual, antes de llamar a `updateRow`, llamaremos a `cancelRowUpdates`
- Para borrar el registro actual, usaremos `deleteRow`. La llamada a este método deja una fila vacía en el `ResultSet`. Si intentamos acceder a ese registro se producirá una excepción
- El método `rowDeleted` nos dice si el registro ha sido eliminado

Restricciones

- Para poder hacer uso de un `ResultSet` arrastable, la sentencia `SELECT` que lo genera debe:
  - Referenciar sólo una tabla
  - No contener una claúsula `join` o `group by`
  - Seleccionar la clave primaria de la tabla
- En el `ResultSet` disponemos de un registro especial, llamado de inserción.
- Nos permite introducir nuevos registros en la tabla

Sentencias de actualización

- La clase `Statement` incorpora un método para realizar actualizaciones: `executeUpdate`
- Recibe una cadena que es la sentencia SQL a ejecutar:
  - `CREATE` (creación de tablas)
  - `INSERT` (inserción de datos)
  - `DELETE` (borrado de datos)
- El método `executeUpdate` devuelve un entero que indica el número de registros afectados (`CREATE` devuelve siempre `0`)

Ejemplos

```
String st_crea = "CREATE TABLE ALUMNOS ( exp
  INTEGER, nombre VARCHAR(32), sexo CHAR(1),
  PRIMARY KEY (exp) )";
  stmt.executeUpdate(st_crea);
String st_inserta = "INSERT INTO ALUMNOS (exp,
  nombre) VALUES(1285, 'Manu', 'M')";
  stmt.executeUpdate(st_inserta);
String st_actualiza = "UPDATE FROM ALUMNOS SET
  sexo = ‘H' WHERE exp = 1285";
  stmt.executeUpdate(st_actualiza);
String st_borra = "DELETE FROM ALUMNOS WHERE exp
  = 1285";
  stmt.executeUpdate(st_borra);
```

Otras llamadas a la BD

- Si no conocemos de antemano el tipo de consulta (la ha introducido el usuario), podemos utilizar el método `execute` de la clase `Statement`.
- El método devuelve un valor booleano, siendo cierto si hay resultados y falso en el caso de una sentencia de actualización
- Si es falso, podemos llamar al método `getUpdateCount` de `Statement` que nos dice el número de registros afectados

Otras llamadas a la BD

- Si hay resultados, los podemos obtener con el método `getResultSet` de `Statement`. Este método devuelve un `ResultSet`
- Si hemos ejecutado un procedimiento en la BD, es posible que tengamos más de un `ResultSet`
- El método `getMoreResult` nos devuelve cierto si existen más resultados. Después de esta llamada podemos volver a llamar a `getResultSet` para conseguir el siguiente resultado

Otras llamadas a la BD

- Si queremos ejecutar varias sentencias SQL a la vez, podemos utilizar el método `executeBatch`
- No permite sentencias de tipo `SELECT`
- Para añadir sentencias usaremos el método `addBatch`
- `executeBatch` devuelve un *array* de enteros indicando el número de registros afectados en cada sentencia

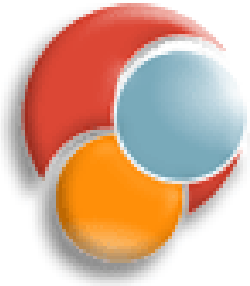
Ejemplo

```
stmt.addBatch("INSERT INTO ALUMNOS(exp,
  nombre) VALUES(1285, 'Manu', 'M')");

stmt.addBatch("INSERT INTO ALUMNOS(exp,
  nombre) VALUES(1299, 'Miguel', 'M')");

int[] res = stmt.executeBatch();
```

Otras llamadas a la BD

- Obtener claves generadas
  - Útil para inserciones en campos autonuméricos

```
ResultSet res =
      sentSQL.getGeneratedKeys();
if(res.next()) {
  id = res.getInt(1);
}
```



¿Preguntas...?